ECEn 425
Real-time and embedded systems

Dr. James Archibald
450P EB

Insight into how we learn

- From Carl Wieman…
- From Derek Muller…
- From Robert Bjork…

What does this mean for us here today?

[illegible]

- [illegible]
- [illegible]
  - [illegible]
  - [illegible]
  - [illegible]
  - [illegible]
  - [illegible]
- [illegible]

What's this class all about?

- An introduction to the design of embedded systems, with emphasis on software
- How are these changing our world?
  - Smart phones and apps
  - Autonomous vehicles
  - Facial recognition systems
  - The Internet of Things
- What role does software play in these devices and systems?

Digital functionality

- Embedded platforms seldom employ fully custom devices;
  - Instead, commodity parts are integrated in a platform-specific way
- System behavior is determined by standard microprocessors customized *through software*
- This brings significant flexibility: to change the function, simply change the program
- But what is the cost of software?

Your ECEn 330 experience

- What did you learn about embedded software from 330?
- How does process of creating software compare with process of creating hardware?
- How does programming compare with traditional engineering tasks, such as constructing a bridge?

Software: the downside

- **Code size** is a problem, memory demands continue to increase
  - Old voice-only cell phones had ~8 MB ROM, 4 MB RAM
  - RAM in iPhone 1: 128MB, iPhone 4: 512MB, iPhone 5/6: 1GB
  - Total code in smart phone: >20 M lines
  - Total code in avg. high end automobile: >100 M lines
  - But productivity just 100-200 lines of code per programmer per month!
    - How can projects ever finish?!
- **Code complexity** is a problem
  - Even small embedded systems can require major development effort
  - Tools used to manage software development have "productivity factors" for real-time embedded systems that are much lower than standard systems

Software: the downside

- Consequence: hardware costs dwarfed by software costs
  - 20:1, 50:1 not uncommon in embedded systems (SW:HW)
- Testing and debugging are responsible for ~50% of the cost of developing conventional software
  - In real-time systems, they are responsible for ~70%
  - Conventional debugging aids don't "see" many of the bugs
  - Generally impossible to exercise all timing possibilities
- Factors contributing to software complexity
  - Typical embedded code uses *concurrent* constructs
  - Target systems can differ greatly from each other
  - Devices often networked (creating security vulnerabilities)

Was your 330 code concurrent?

Software dominates

- Strong demand for engineers developing embedded systems.
- Most common title is software engineer
  - Deceptive: means programmer that uses scope, logic analyzer to debug!
- Most common background:
  - CpE or EE
  - Why not CS?
    - SW drives HW directly
    - Detailed knowledge of HW required by SW developer
    - Often no safety net: some SW bugs can fry hardware

Embedded software

- Embedded systems are important part of computer engineering at BYU
- We have other classes that emphasize the hardware.
- Emphasis in this class is the software
  - What is unusual about software for embedded systems?
  - What are challenges involved in designing, coding, and debugging this type of software?
- You'll be able to answer these questions in detail by end of semester!

Class emphasis

- Main focus is the lab sequence
  - You will create a simple but complete RTOS
    - RTOS = real-time operating system
    - Something similar used in many embedded systems
    - Development done in user-friendly simulation environment
  - Goals:
    - Understand software structures, algorithms required for preemptive multi-tasking
    - Understand relationship of hardware interrupts and software interrupt handlers
    - Understand how to construct software to respond to important events in a timely fashion – critical in real-time systems

My goals

- I want students in this class to
  - become interested in the challenges of designing reliable real-time and/or embedded systems
  - know enough about embedded software development to do well in job interviews in that discipline
  - have a thorough knowledge of the functionality and limitations of real-time operating systems
  - gain experience creating application code with real-time constraints
  - be better computer engineers by having greater understanding of the underlying system and acquiring more design, programming, and debugging experience

Labs

- Initial labs done by each individual; RTOS labs done in groups of two.
  - After lab 4, working alone requires special permission.
  - Significant benefits of having someone to ~~argue~~ work with!
- Not color-by-number projects:
  - The API is specified (system functions you code up)
  - You design the underlying data structures, use them in a consistent way, and make it all work
- Projects can be time consuming
  - Debugging concurrent code is challenging
  - Unlike virtually all previous coding you've done

Why challenging?

- They are <u>not</u> artificially complicated; the complexity is inherent
- The challenge is <u>not</u> creating lots of code
  - Size of my full kernel (including comments, blank lines)
    - C code (.c and .h): 955 lines total in 4 files
    - Assembly code (.s): 175 lines total in 2 files
- The challenge is that your code is concurrent and must work reliably; the details really matter
- Warning:
  - Students report ~10x difference in time to do labs
  - Example: one group takes 2 hours, another takes 20

Lab survival skills

- <u>Average</u> time invested is not excessive for a 4 credit hour ECEn class.
- How to avoid getting on the high end of the distribution:
  - Work through all the details in the design phase
  - Plan from the outset how you will test your code
  - Include functions for testing in your design
  - Think every code change through carefully and test thoroughly
- An extra hour spent in **design** can reduce **debugging** by 10x

> 8 hours of programming can save you 10 minutes with pencil and paper.
> Mike Goodrich

Prerequisites

- I'm assuming you've taken (and remember things from!):
  - CS 142
  - ECEn 220
  - ECEn 330
- Critical material:
  - Knowledge of C programming language
  - Experience designing and debugging code with timing constraints
  - Understanding general operation of computer systems

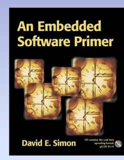
Lab policies

- Keep up!
  - Labs build on each other; hard to get caught up if you get behind
  - Additional motivation: late lab penalty
    - 25% per day first two weekdays
    - 20% per weekday thereafter, to max of 90% off
  - Department policy: <u>complete all labs to pass the class</u>
- Honor code expectation:
  - All code you use must be original (written by you + partner)

Homework

- Combination of two kinds of assignments
  - Excursions relating to C
  - Selected problems from text
- Upload file (.txt or .pdf) to Learning Suite before 11:00pm on due date
  - Please keep submissions neat and organized
  - Used fixed-point font for code listings, with proper indentation
- Assignments can be accessed from the class web page

Text, Attendance



- An Embedded Software Primer, Simon
  - Very readable and technically sound
  - Written by experienced practitioner.
- Class attendance is important
  - We will discuss many things not covered in the text:
    - Important issues related to labs – insight that will save you time!
    - Discussion of case studies, other supplemental material
    - Something interesting to start every class

Grading

- Overall grade determined by:
  - 30% from labs
  - 10% from homework
  - 30% from midterms (two, closed book, in class)
  - 30% from final (closed book, at scheduled time)
- Midterms & solutions from Fall 2018 are online
- Letter grades assigned subject to college/dept. guidelines
  - Class GPA will be ~3.1; median grade will be a B

Miscellaneous

- Read the syllabus and other online class material.
  - Syllabus is on Learning Suite, assignments on webpage
  - Note particularly the online schedule:
    - 1st HW due on Learning Suite before 11:00 PM next Tuesday
    - 1st lab due next Thursday: introduction to class tools (passed off to TA; send email if no TA on duty, pass off later)
- Questions?

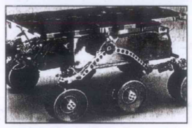
Let's go!

Thought experiment

- Your team is designing a rover to explore the surface of Mars.
- What special challenges must your team address?
- What should the onboard control software do if something goes wrong?

Bug of the Month

Man Finds Bugs on Mars



Pathfinder bugs inhibited the Rover.

PHOTOGRAPH: NASA/JPL/CALTECH © 1997

Wherever a computer goes, bugs are sure to follow. When the Mars Pathfinder developed a glitch, NASA had to somehow upload new code without losing valuable time needed for exploration. The most confounding bug on the Pathfinder mission appeared July 10. Steven Stolper, software engineer for the Mars Pathfinder, calls it "one in a million, insidious, and hard to replicate." The snafu arose because the OS, Wind River's VxWorks, developed a mutual-exclusion problem: A low-priority function (in this case, recording weather) interfered with the system's multitasking schedule. The system couldn't finish all the tasks it needed to, missed a real-time deadline, and then shut itself down. "It's a kind of interplanetary Control-Alt-Delete," says Stolper. "When things go wrong, the system goes into a power-safe mode and waits for ground control to help out." Without a fix being implemented, this problem would replay itself over and over.

To identify the bug, engineers recreated the malfunction on Earth, identified the offending subroutine, and uploaded the binary difference between the new code and the buggy code on the Pathfinder. –Jason Krause

Send yours to jkrause@mgh.com!

OCTOBER 1997 BYTE 23

Nuggets from the preface

"Perversities of embedded systems"

- Inconsistent terminology between companies
  - We'll try hard to be consistent with book
- Incredible diversity of embedded systems
  - From 8-bit microcontrollers with only internal memory to 32-bit machines with gigabytes of external memory
  - Code size: from under 500 bytes to many megabytes

Perversities, cont.

> "Any rule followed by 85% of engineers as part of the accepted gospel of standard practice has to be broken by the other 15% just to get their systems to work."

If this is true, what role should rules play?

The C language

- The *lingua franca* of embedded systems
  - C compiler available even for small microcontrollers
  - C is simple; resulting code and behavior predictable
  - C++, Java more complicated; less frequently supported
- C essential in this class
  - Labs done in C and assembly
  - Text assumes reading knowledge (C!!)
- C is reasonably portable, but far from perfect

> C combines the power of assembly language with the ease of use of assembly language.
> Mark Pearce

"Hungarian" variable naming

- Used in examples in book.
- Variables have type-specific prefix:
  - by = unsigned char
  - i = integer
  - p_i = pointer to an integer
  - f = flag
  - a_ = array of ...
- What benefits might this convention offer?
  - Consider similar approach in your own code.

µC-OS

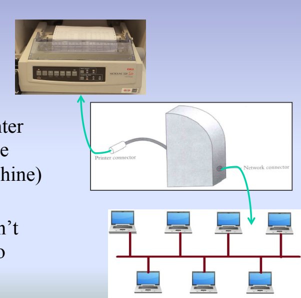
- A *microkernel* or *RTOS* on accompanying CD
  - A small, streamlined OS for embedded systems
  - Functionality is similar to the RTOS you will code up
  - µC-OS not shareware but can be used for educational purposes
  - Has been used in our department for some senior projects
- You may use the CD but are not required to do so
  - You are free to browse or study the source code
    - Many aspects of kernel are similar to ours, but there are also substantive differences
  - For your convenience, contents have been copied onto the department file-server. (See syllabus for details.)

Chapter 1

Embedded systems software

- Must deal with a different set of issues than typical desktop or enterprise software
- It must...
  - handle situations that don't arise in conventional software.
  - do several things at once.
  - respond in timely way to external events (button presses, sensor readings, etc.).
  - cope with unusual conditions without human intervention.
  - meet strict processing deadlines.
  - never fail. (Is this really possible?)

"Telegraph" example



- Connects an old style printer (designed to connect to the serial port of a single machine) to a network
- Lots of complications; can't just copy incoming data to other side

- Complications:
  - Network data is broken into packets which may arrive out-of-order, or be lost entirely.
  - Multiple machines may try to use the printer at same time.
  - Printer status must be made available at all times to any requesting computer regardless of what printer is doing.
  - Must work with different kinds of computers without any special customization.
  - Must figure out type of attached printer at power up.
  - Must provide response to certain network packets within 200 ms.
  - Must handle timeouts – if computer crashes while printing, job must be terminated, printer reset, and next job started.
  - Must work without human intervention.

Other design issues

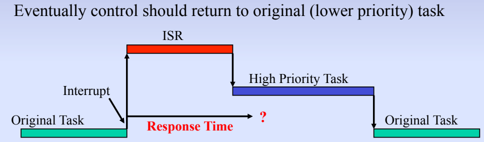
- Throughput
  - Is it fast enough to keep up with data rates of transfers from computer to printer?
- Response time
  - Can it provide a timely response to important events?
- Testability
  - Can it be shown to work under all conditions?
- Debugability
  - How will errors in system be located and fixed?
- Reliability
  - Will it work as well as customers expect?

Other system design concerns

- Power consumption
  - How do we design to maximize battery lifetime?
  - Software can power-down system when unused, but how to turn it on again?
- Minimizing system cost
  - Saving a few cents is a big deal (assuming high volume, low margins)
  - Minimize size of ROM and RAM; use small OS
- Balancing conflicting needs
  - Substantial computation vs. fast response time
  - Response time is best when system isn't otherwise busy

Achieving acceptable response times

- Designer creates separate "tasks" (C functions) with unique priorities
- System runs highest priority ready task at each instant
- Interrupt code can make another task ready; can run on ISR return
- Eventually control should return to original (lower priority) task



How does this impact the perceived responsiveness of the system?
What are design tradeoffs? (Did our 330 code work this way?)
What system support is needed to make this work?

Choosing an embedded processor

- Commodity parts: wide range of models are available
  - Differ in cost, performance, features
- How can companies offer so many choices?
  - Enormous volume
    - In 2015, 15 billion ARM processors were sold
    - This is more processors than Intel has sold in its corporate history!
  - Proven technology
    - Mature processes, small dies with high yield and low cost

Microprocessor vs. *microcontroller*?

Embedded systems

- Often defined in terms of what they don't have:
  - Keyboards – but may have buttons or keypads
  - Screens – but may have LEDs, small LCD
  - Disk drives – but may use *flash* in similar way
  - CD players, modems – but may have network connections

Discussion

- If there is no disk, how does a program get loaded at power-up?
- Do embedded systems require both ROM and RAM?
- How are embedded systems debugged without a keyboard and monitor?
- What are the “tasks” in our earlier example?
  - How are they represented?
  - Who decides how many and assigns the priorities?
- Lots more on these issues!

Schedule of Topics

- That’s it for Chapter 1
  - Think about the implications of what we’ve talked about
- Next up:
  - Background information on the x86 architecture and tools that we’ll be using this semester

The 8086 architecture

> The x86 isn’t all that complex – it just doesn’t make a lot of sense.
>
> Mike Johnson
> Leader of x86 Design at AMD

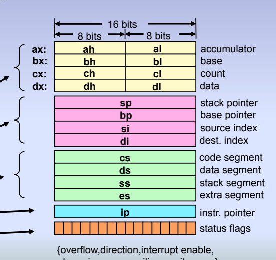
> Whatever the artistic failures of the 80x86, keep in mind that there are more instances of this architectural family than of any other in the world… Nevertheless, its checkered ancestry has led to an architecture that is difficult to explain and impossible to love.
>
> John Hennessy and David Patterson
> Computer Architecture: A Quantitative Approach

x86 history

- 1978: 16-bit 8086 announced, assembly-language compatible with 8-bit 8080. Many registers added – 8080 was accumulator machine.
- 1980: 8087 FP coprocessor announced, 60 FP instructions added that use an extended stack architecture.
- 1982: 80286 extended address space to 24 bits; maintained compatibility with 8086 *real addressing*.
- 1985: 80386 extended architecture to 32 bits, with 32-bit registers and address space.
- Many extensions and add-ons since.

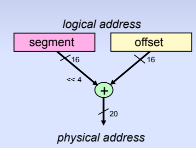
8086 registers

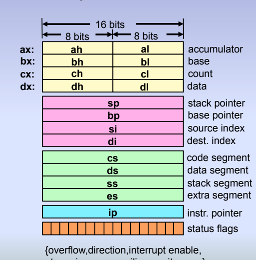
- Both 8-bit and 16-bit operations and registers
- Registers have specific purposes:
  - general purpose
  - pointer and index
  - segment
  - instruction pointer
  - flags



Accessing memory

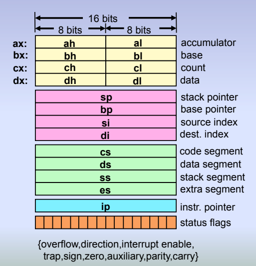
- Memory addressing:





8086 segments

- Memory segments
  - can start on any 16-byte boundary
  - can be as small as 16 bytes, as large as 64 KB
  - can overlap: a memory byte can be referenced with multiple logical addresses



Using segment registers

- Segment registers exist for
  - code
  - data
  - stack
  - "extra"
- Memory accesses use current segment register contents – usually implicit
  - Can specify another segment register with explicit override
  - Increased overhead to access data in another segment
  - Increased overhead to call function in another segment
  - Simplification for 425: all code and data (RTOS + application) should fit in one 64 KB segment. All segment registers should be zero.

Accessing operands

- Operands can be in registers, immediate data, or memory
  - Assembly convention: first operand is both a source and destination
  - Examples:

```
add  ax,bx                        ; 16-bit add, register operands, result in ax
add  ah,cl                        ; 8-bit add, register operands
add  bx, 3                        ; 16-bit, register and immediate operands
add  word [bx], 3                 ; bx holds ptr, must specify size of op
add  word [bx], word [si]         ; illegal! one memory addr per instruction, max
mov  ax, [si+2]                   ; reg + constant to specify address
mov  ax, [constant+basereg+indexreg]     ; most general form
```


8086: a 16-bit CPU

- Size of C data types with our compiler on 8086:

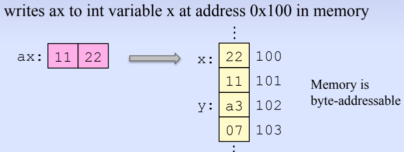
char	8 bits
enum	16 bits
short int	16 bits
int	16 bits
long	32 bits

pointers are a bit more complicated (Can you guess why?)

- 32-bit operations can require many instructions
  - Avoid whenever possible: avoid performance penalty
- No floating-point operations

8086: Little-endian

- Least significant byte of value is stored first (at low address)
  - Example: assume 16-bit value 0x1122 is in register ax
  - CPU writes ax to int variable x at address 0x100 in memory



  - Biggest problem for us: multi-byte value appears backwards when memory contents listed in order

8086: Assembly notes

- At most one memory address allowed per instruction
- At most one immediate value allowed per instruction
- Generally, operands must be of the same type
- NASM (The Netwide Assembler)
  - Labels are case-sensitive
  - Instruction and register names are not case-sensitive

8086: Flags

- Only 9 of 16 bits are used
  - Referred to as of, df, if, tf, sf, zf, af, pf, cf
- Set to reflect the result of various instructions and operations
  - Example: cf set if result generated a carry; adc (add with carry) used to do 32-bit adds with 16-bit operations
  - Example: of, sf, zf, cf set by arithmetic operations and comparisons; values determine outcome of conditional jumps
- Class webpages on instruction set give operational details

8086: Multiplication and division

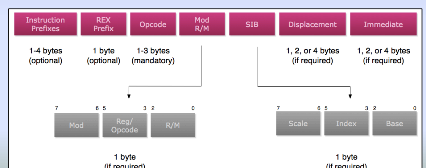
- Require extra care to perform properly
- Multiplying two 16-bit operands produces a 32-bit result, overwriting two 16-bit registers
  - Operands must be in correct registers
  - Answer must be extracted from the right registers
- Part of Lab 1: figure out how these instructions work, use them in assembly code
  - Good way to proceed: write simple C programs, compile them (using class tools), and study assembly output
  - Then write assembly code that does just what you want
  - Don't be afraid to experiment!

Assembly vs. C

- Why program in assembly?
  - Performance: your code may be better than compiler's code
    - This is **a lot** of work and seldom worth it
    - Our compiler is not great, but good enough for this class
  - Execute specific system instructions, or access specific registers
    - Examples: enable/disable interrupts, save/restore register context
- **Strongly recommended**: minimize your assembly code
  - Use C for everything you can; use assembly only when necessary
  - Inline assembly is dangerous – "prohibited" in this class

8086: Instruction encoding

- "Saving the worst for last, the encoding of instructions in the 8086 is complex, with many different instruction formats." (Hennessy & Patterson, p. D-11)
- Really doesn't affect us, but you should be grateful that you don't have to write the simulator or build a hardware decoder!



8086: Parting shot

> The complexity of the x86 is not an impassable barrier… The biggest weakness in the x86 instruction set is the lack of registers coupled with an extremely painful addressing scheme.
>
> Mike Johnson
> Leader of x86 Design at AMD

Lab1 assignment

- Use the class tools (compiler, assembler, simulator) to compile and run a simple program
- Write the 8086 assembly code (fleshing out a skeletal function) to compute the expression below in 20 instructions or less
  - gvar+((a*(b+c))/(d-e))
- Requires access to global variable (int gvar) and incoming parameters (int a, char b, char c, int d, int e) on the stack
  - See lab info webpages for useful info about stack, assembly syntax

Using the toolset

- How to create and run a program
  - Edit file, say `ex1.c`
  - Run C preprocessor on file (`cpp ex1.c ex1.i`)
  - Compile (`c86 -g ex1.i ex1.s`)
  - Add clib.s code (`cat clib.s ex1.s > ex1fin.s`)
  - Assemble file (`nasm ex1fin.s -o ex1.bin -l ex1.lst`)
  - Run simulator (`emu86`) (new window pops up)
  - Load executable (`l ex1.bin`)
  - Run program in simulator (`e`)
- You will quickly realize the benefits of using `make`!

©J Archibald 425 F19 1:55

Simulator functionality

- Has lots of features that can save you time!

Quick Command Lookup

?	help	m	mem monitor info
a	assert	ma	mem monitor (access)
addr	address lookup	mb	mem monitor (become)
b	breakpoint	mm	mem monitor (modify)
c	clear breakpoint	mr	mem monitor (read)
clear	clear breakpoints & monitors	mw	mem monitor (write)
Ctrl+b	program break	n	clear mem monitor
d	disassemble	o	step over
ds	dump stack	pb	poke byte
dump	dump	pd	poke double
dumpb	dump byte	pr	poke register
dumpw	dump word	pw	poke word
dumpd	dump dword	q	quit
e	execute	r	print regs (hex)
eto	execute to	rd	print reg (signed dec)
f	reg monitor info	ru	print reg (unsigned dec)
fb	reg monitor (become)	regs	register info
fm	reg monitor (modify)	s	step
g	clear reg monitor	simptris	simptris mode
h	help	t	tick
hist	inst/int history	v	verbose
l	load	w	wipe output window
lexe	load exe		

©J Archibald 425 F19 1:56

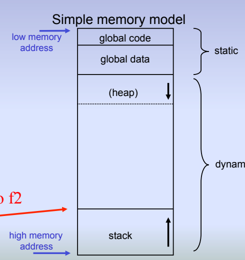
Stack essentials

- Consider execution of this C code:

```
void f1 (void)
{
    ...
}

void f2 (void)
{
    ...
    f1();
}

main ()
{
    ...
    f2();
}
```

just before call to f2

sp



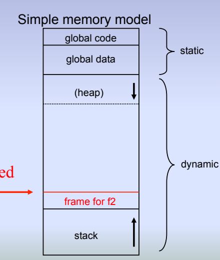
©J Archibald 425 F19 1:57

Stack essentials

- Consider execution of this C code:

```
void f1 (void)
{
    ...
}

void f2 (void)
{
    ...
    f1();
}

main ()
{
    ...
    f2();
}
```

then f2 is called,
new stack frame created
sp



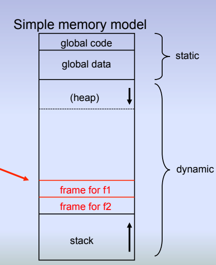
©J Archibald 425 F19 1:58

Stack essentials

- Consider execution of this C code:

```
void f1 (void)
{
    ...
}

void f2 (void)
{
    ...
    f1();
}

main ()
{
    ...
    f2();
}
```

then f1 is called,
new frame created
sp



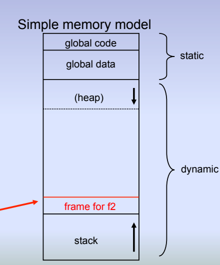
©J Archibald 425 F19 1:59

Stack essentials

- Consider execution of this C code:

```
void f1 (void)
{
    ...
}

void f2 (void)
{
    ...
    f1();
}

main ()
{
    ...
    f2();
}
```

f1 returns,
f1 frame “removed”
sp



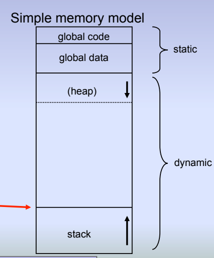
©J Archibald 425 F19 1:60

Stack essentials

- Consider execution of this C code:

```
void f1 (void)
{
    ...
}

void f2 (void)
{
    ...
    f1();
}

main ()
{
    ...
    f2();
}
```

f2 returns,
f2 frame "removed"



Is stack exactly as it was before call to f2()?

8086 Tools: Example 1

```
/* ex1.c */
void printInt(int result);

void main(void)
{
    int x = 10;
    int y = 3;
    int result;

    result = x / y;
    printInt(result);
}
```

```
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
        jmp     main    ; Jump to program start
        align   2
main:
        ; >>>>> Line: 5     > {
        jmp     L_ex1_1
L_ex1_2:
        ; >>>>> Line: 10    > result = x / y;
        mov     word [bp-2], 10
        mov     word [bp-4], 3
        ; >>>>> Line: 10    > result = x / y;
        mov     ax, word [bp-2]
        cwd
        mov     cx, word [bp-4]
        idiv    cx
        mov     word [bp-6], ax
        ; >>>>> Line: 11    > printInt(result);
        push    word [bp-6]
        call    printInt
        add     sp, 2
        mov     sp, bp
        pop     bp
        ret
L_ex1_1:
        push    bp
        mov     bp, sp
        sub     sp, 6
        jmp     L_ex1_2
```

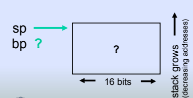

Example 1 notes

- Stack details:
  - bp saved (why?), then set to value of sp, new value used throughout
  - bp used in most stack references (locals, args)
  - sp modified frequently: `push`, `pop`, `call`, `ret`
    - `call` pushes IP onto stack, sets IP to addr
    - `ret` pops word from stack, puts it in IP
  - sp at end will have original value
- Instruction details:
  - `cwd`: convert word to doubleword; sign extends ax to fill dx, result in dx::ax. (Use of ax, dx is implicit.)
  - `idiv`: dx::ax / cx, result in ax, remainder in dx. (Use of ax, dx is implicit.)

Example 1


```
void main(void)
{
    int x = 10;
    int y = 3;
    int result;

    result = x / y;
    printInt(result);
}
```

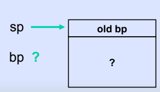
```
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
        jmp     main    ; Jump to program start
        align   2
main:
        ; >>>>> Line: 5     > {
        jmp     L_ex1_1
L_ex1_2:
        ; >>>>> Line: 10    > result = x / y;
        mov     word [bp-2], 10
        mov     word [bp-4], 3
        ; >>>>> Line: 10    > result = x / y;
        mov     ax, word [bp-2]
        cwd
        mov     cx, word [bp-4]
        idiv    cx
        mov     word [bp-6], ax
        ; >>>>> Line: 11    > printInt(result);
        push    word [bp-6]
        call    printInt
        add     sp, 2
        mov     sp, bp
        pop     bp
        ret
L_ex1_1:
        push    bp
        mov     bp, sp
        sub     sp, 6
        jmp     L_ex1_2
```

Example 1


```
void main(void)
{
    int x = 10;
    int y = 3;
    int result;

    result = x / y;
    printInt(result);
}
```

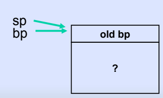
```
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
        jmp     main    ; Jump to program start
        align   2
main:
        ; >>>>> Line: 5     > {
        jmp     L_ex1_1
L_ex1_2:
        ; >>>>> Line: 10    > result = x / y;
        mov     word [bp-2], 10
        mov     word [bp-4], 3
        ; >>>>> Line: 10    > result = x / y;
        mov     ax, word [bp-2]
        cwd
        mov     cx, word [bp-4]
        idiv    cx
        mov     word [bp-6], ax
        ; >>>>> Line: 11    > printInt(result);
        push    word [bp-6]
        call    printInt
        add     sp, 2
        mov     sp, bp
        pop     bp
        ret
L_ex1_1:
        push    bp
        mov     bp, sp
        sub     sp, 6
        jmp     L_ex1_2
```

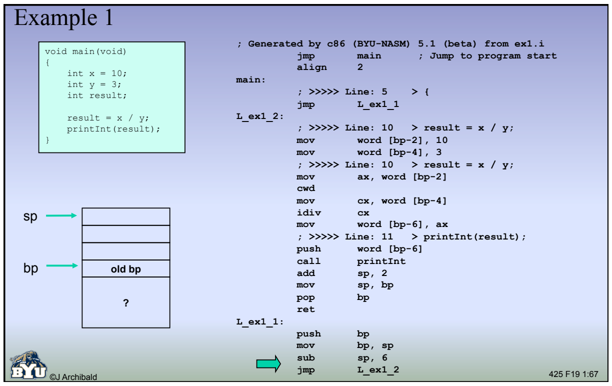



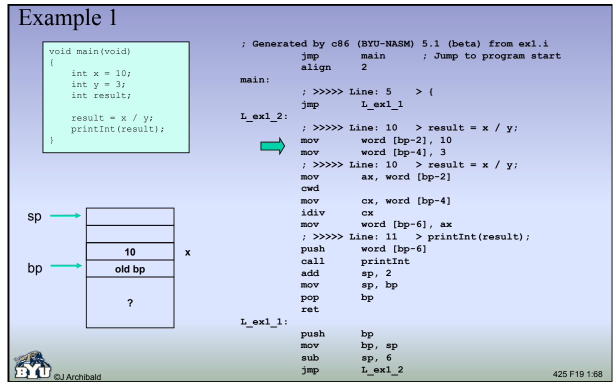
Example 1

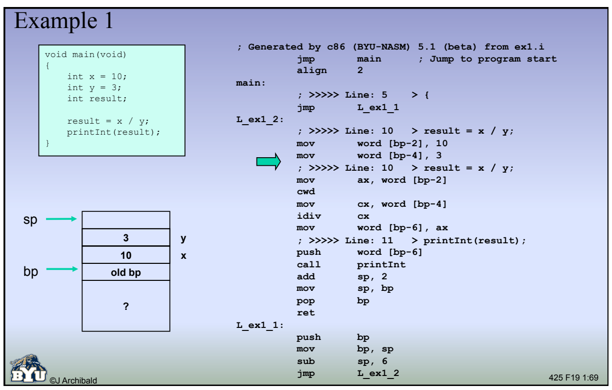

```
void main(void)
{
    int x = 10;
    int y = 3;
    int result;

    result = x / y;
    printInt(result);
}
```

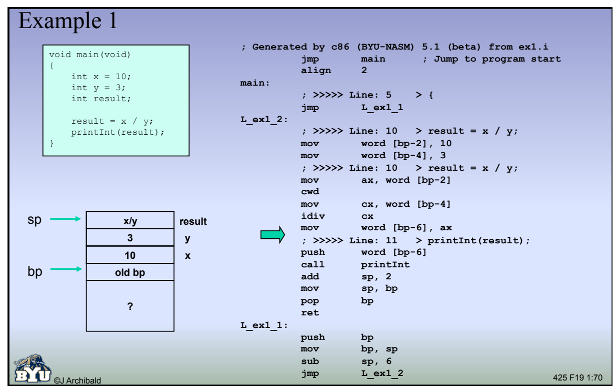
```
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
        jmp     main    ; Jump to program start
        align   2
main:
        ; >>>>> Line: 5     > {
        jmp     L_ex1_1
L_ex1_2:
        ; >>>>> Line: 10    > result = x / y;
        mov     word [bp-2], 10
        mov     word [bp-4], 3
        ; >>>>> Line: 10    > result = x / y;
        mov     ax, word [bp-2]
        cwd
        mov     cx, word [bp-4]
        idiv    cx
        mov     word [bp-6], ax
        ; >>>>> Line: 11    > printInt(result);
        push    word [bp-6]
        call    printInt
        add     sp, 2
        mov     sp, bp
        pop     bp
        ret
L_ex1_1:
        push    bp
        mov     bp, sp
        sub     sp, 6
        jmp     L_ex1_2
```

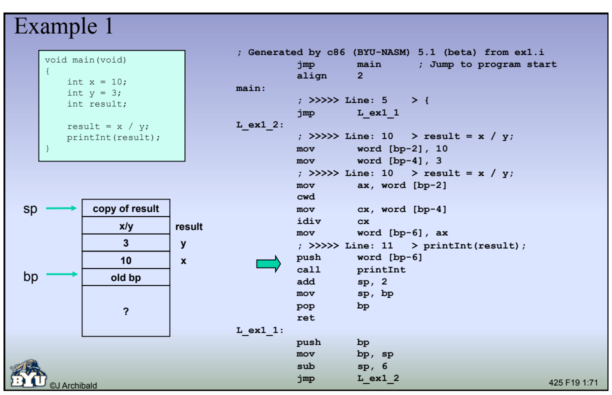



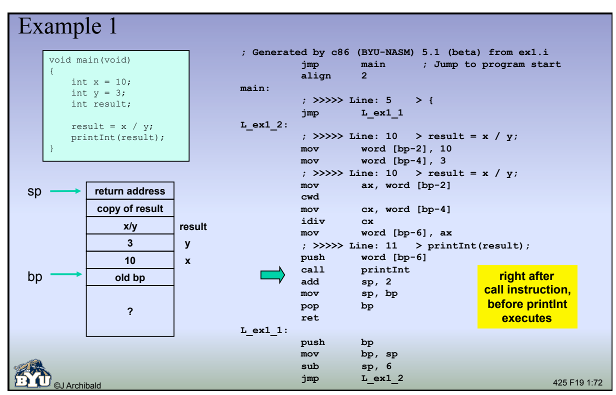

Example 1
void main(void)
{
int x = 10;
int y = 3;
int result;
result = x / y;
printInt(result);
}
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
sp
bp
old bp
?
©J Archibald
425 F19 1:67

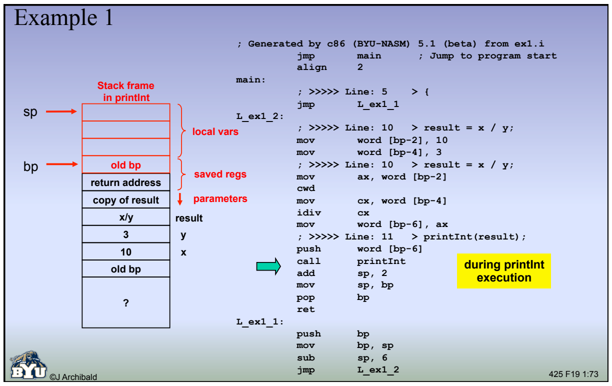

Example 1
void main(void)
{
int x = 10;
int y = 3;
int result;
result = x / y;
printInt(result);
}
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
sp
bp
10
x
old bp
?
©J Archibald
425 F19 1:68

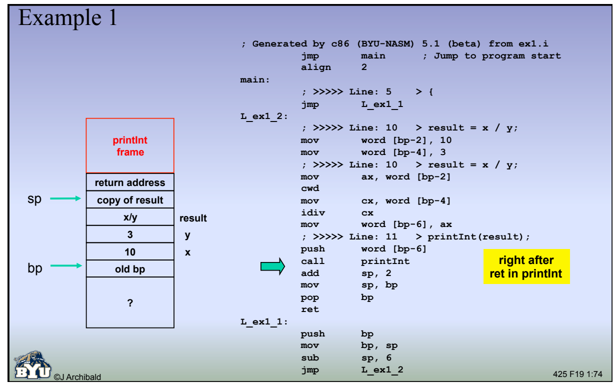

Example 1
void main(void)
{
int x = 10;
int y = 3;
int result;
result = x / y;
printInt(result);
}
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
sp
bp
3
y
10
x
old bp
?
©J Archibald
425 F19 1:69

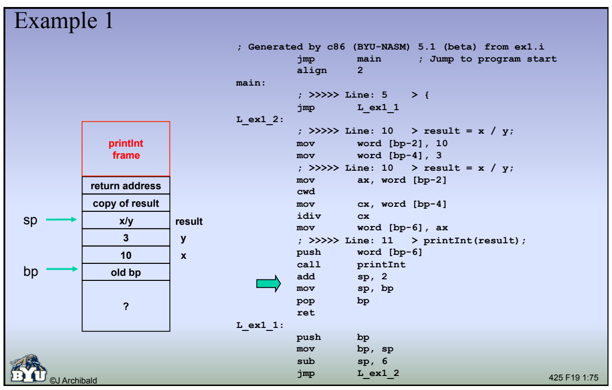

Example 1
void main(void)
{
int x = 10;
int y = 3;
int result;
result = x / y;
printInt(result);
}
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
sp
bp
x/y
result
3
y
10
x
old bp
?
©J Archibald
425 F19 1:70

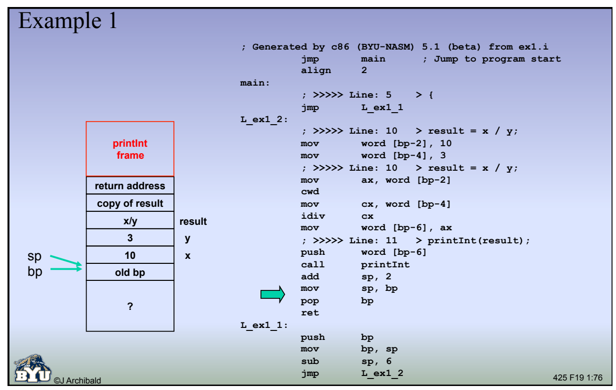

Example 1
void main(void)
{
int x = 10;
int y = 3;
int result;
result = x / y;
printInt(result);
}
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
sp
bp
copy of result
x/y
result
3
y
10
x
old bp
?
©J Archibald
425 F19 1:71

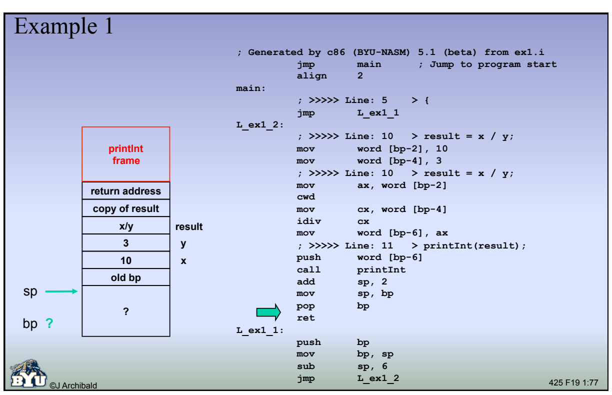

Example 1
void main(void)
{
int x = 10;
int y = 3;
int result;
result = x / y;
printInt(result);
}
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
sp
bp
return address
copy of result
x/y
result
3
y
10
x
old bp
?
right after call instruction, before printInt executes
©J Archibald
425 F19 1:72

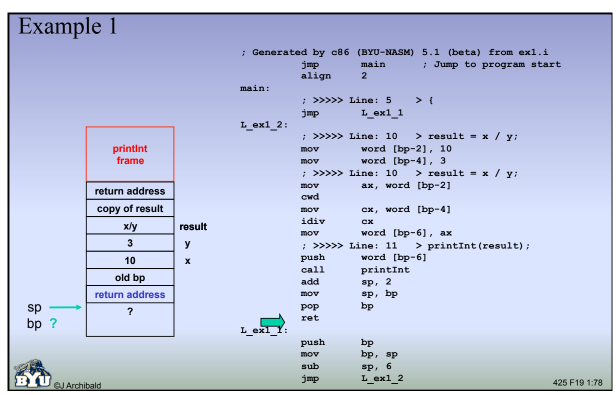

Example 1
Stack frame in printInt
sp
bp
old bp
return address
copy of result
x/y
3
10
old bp
?
local vars
saved regs
parameters
result
y
x
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
during printInt execution
©J Archibald
425 F19 1:73


Example 1
printInt frame
return address
copy of result
x/y
3
10
old bp
?
sp
bp
result
y
x
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
right after ret in printInt
©J Archibald
425 F19 1:74


Example 1
printInt frame
return address
copy of result
x/y
3
10
old bp
?
sp
bp
result
y
x
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
©J Archibald
425 F19 1:75


Example 1
printInt frame
return address
copy of result
x/y
3
10
old bp
?
sp
bp
result
y
x
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
©J Archibald
425 F19 1:76


Example 1
printInt frame
return address
copy of result
x/y
3
10
old bp
?
sp
bp ?
result
y
x
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
©J Archibald
425 F19 1:77


Example 1
printInt frame
return address
copy of result
x/y
3
10
old bp
return address
?
sp
bp ?
result
y
x
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex1.i
jmp main ; Jump to program start
align 2
main:
; >>>>> Line: 5 > {
jmp L_ex1_1
L_ex1_2:
; >>>>> Line: 10 > result = x / y;
mov word [bp-2], 10
mov word [bp-4], 3
; >>>>> Line: 10 > result = x / y;
mov ax, word [bp-2]
cwd
mov cx, word [bp-4]
idiv cx
mov word [bp-6], ax
; >>>>> Line: 11 > printInt(result);
push word [bp-6]
call printInt
add sp, 2
mov sp, bp
pop bp
ret
L_ex1_1:
push bp
mov bp, sp
sub sp, 6
jmp L_ex1_2
©J Archibald
425 F19 1:78

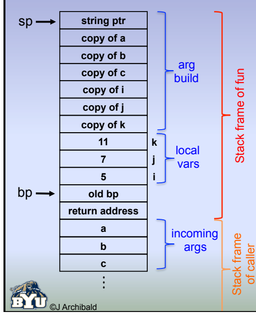
8086 Tools: Example 2

```c
/* ex2.c */

int printf();

void fun(int a, int b, int c)
{
   int i, j, k;
   i = 5;
   j = 7;
   k = 11;

   printf("%d %d %d %d %d %d",
      a, b, c, i, j, k);
}
```

Note: printf is not available in clib.s. Used for illustration only.

```
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex2.i
        jmp     main    ; Jump to program start
L_ex2_1:
        db      "%d %d %d %d %d %d",0
        align   2
fun:
        ; >>>>> Line: 8    > {
        jmp     L_ex2_2
L_ex2_3:
        ; >>>>> Line: 10   > i = 5;
        mov     word [bp-2], 5
        ; >>>>> Line: 11   > j = 7;
        mov     word [bp-4], 7
        ; >>>>> Line: 12   > k = 11;
        mov     word [bp-6], 11
        ; >>>>> Line: 14   > printf("%d %d %d %d..
        push    word [bp-6]
        push    word [bp-4]
        push    word [bp-2]
        push    word [bp+8]
        push    word [bp+6]
        push    word [bp+4]
        mov     ax, L_ex2_1
        push    ax
        call    printf
        add     sp, 14
        mov     sp, bp
        pop     bp
        ret
L_ex2_2:
        push    bp
        mov     bp, sp
        sub     sp, 6
        jmp     L_ex2_3
```


Example 2




```
; Generated by c86 (BYU-NASM) 5.1 (beta) from ex2.i
        jmp     main    ; Jump to program start
L_ex2_1:
        db      "%d %d %d %d %d %d",0
        align   2
fun:
        ; >>>>> Line: 8    > {
        jmp     L_ex2_2
L_ex2_3:
        ; >>>>> Line: 10   > i = 5;
        mov     word [bp-2], 5
        ; >>>>> Line: 11   > j = 7;
        mov     word [bp-4], 7
        ; >>>>> Line: 12   > k = 11;
        mov     word [bp-6], 11
        ; >>>>> Line: 14   > printf("%d %d %d %d..
        push    word [bp-6]
        push    word [bp-4]
        push    word [bp-2]
        push    word [bp+8]
        push    word [bp+6]
        push    word [bp+4]
        mov     ax, L_ex2_1
        push    ax
        call    printf
        add     sp, 14
        mov     sp, bp
        pop     bp
        ret
L_ex2_2:
        push    bp
        mov     bp, sp
        sub     sp, 6
        jmp     L_ex2_3
```


Example 2: discussion

- Note parameter conventions:
  - incoming parameters at top of previous stack frame
  - outgoing parameters placed at top of current frame
  - C parameter order (left to right) matches stack order (top to bottom)
- Questions
  - Why is sp decremented by 6 (as in ex1.s)?
  - Why is sp incremented by 14 just before resetting?
  - Can local variables be stored in registers?
  - How would function return a value?
    - By convention: byte in al, word in ax, dword in dx::ax

Stack frames

- What do they hold besides local variables and parameters?
  - Register values: some must be saved (and later restored) if they are used
  - Compilers assign registers to one of two categories:
    - caller-saved: called function can overwrite without saving and restoring
      - Examples: ax, cx, dx (based on C86 code examples)
    - callee-saved: caller can assume it will retain its value over call
      - Examples: bp
- Note that push always writes 16-bit value
  - Full 16 bits used for byte-sized args, local vars on stack
    - Least significant byte for args, most significant byte for local vars (?!)
    - See "C Calling Convention" web page – under lab info
  - Ensures that all memory accesses to stack are *aligned*

What's important here?

- Every C compiler has its own conventions on each CPU
- If you want to write assembly code that works with C code, you must observe compiler conventions
  - for incoming parameters
  - for outgoing parameters
  - for return values
  - for registers: callee- and caller-saved
  - for creating and removing stack frames
- Lab 2 asks questions about program details:
  - addr for breakpoint, addr of global, addr of local, initial value of sp, largest stack size, instruction encoding, etc.
- **Read and reread documentation on lab webpages!**